Advanced Expression
Template Concepts

Christoph Pflaum

work with Alexander Linke and Jochen Härdtlein

High Performance Computing Group
School of Advanced Optical Technologies (SAOT)
at the University Erlangen-Nürnberg
Purpose of Simulation Software

- general multi-physics software
- simulation software for a specific application (e.g. simulation of statics of houses)
- software for one or group of users
- library for numerical simulation
- test code for an algorithm or a single application (no reuse of the software)
Simulation Projects of Optical Waves at the LSS

- wave guides for optical metrology
  (cooperation with Laser Laboratory Göttingen, LLG)

- simulation of laser resonators
  (cooperation with LASCAD, Infineon, …)

- solar cell simulation
  (cooperation with Institute of Energy Research, Jülich)

- …
Simulation Software of the HPC Group in Erlangen

Expression Template Libraries
- FE on block structured grids
- FD on staggered grids
- FE on special semi-unstructured grids
- FE local stiffness matrices

Applications:
- test codes for iterative solvers
- simulation of semiconductor lasers
- simulation of solid state lasers in the commercial code LASCAD
- solar cell simulations wave guide simulations
Expression Templates

Aims of Expression Templates:

- Implement algorithms in a language which is close to the mathematical language!
- Obtain optimal efficiency by template constructions!

(Expression templates were invented independent from each other by T. Veldhuizen and D. Vandevoorde.)
Expression Templates

Basic expressions in a mathematical language are:

\[ d = a + b; \]
\[ d = a + b + c; \]

How can these expressions be implemented in an efficient way, if \( a, b, c, d \) are large vectors?


```cpp
class vector  {
private:
    int length;
    double*  v;
public:
    double get(int i) {  return v[i]; }...
};

vector  operator + ( vector& a,  vector& b) {   ...
    help = new double(length);
    for(int i=0;i<length;++i) {
        help[i] = a.get(i) + b.get(i);
    }
    return vector(help);
}
```
Inefficient Implementation

For each operator + in the expression

\[ d = a + b + c; \]

- one iteration is performed and
- one auxiliary vector is allocated.
Efficient Implementation of +

class sum_vector
{
private:
    double *va, *vb;
public:
    double get(int i) const { return va[i] + vb[i]; }
};

Sum_vector operator + (const vector& a, const vector& b) { ... };

class vector
{
public:
    void operator=(const sum_vector& sum) { for(int i=0; i<length; ++i) v[i] = sum.get(i); }
};
The expression
\[ d = a + b; \]
- is evaluated by one iteration and
- without an auxiliary vector.

Expression templates are needed for general expressions like
\[ d = a + b + c; \]
Classical Expression Templates

template<class A, class B>
class Sum {
    A va;
    B vb;
    public:
        double get(int i) const {
            return va.get(i)+vb.get(i);
        }
};

template <class A, class B>
    Expr<Sum<Expr<A>,Expr<B> > >
operator+ (const Expr<A>& a, const Expr<B>& b){
    return Expr< Sum<Expr<A>,Expr<B> > > (Sum<Expr<A>,Expr<B> > (a,b) );
};
The evaluation of an expression is performed by the operator = :

```cpp
class vector {   ...
   public:
       template < class Right_Expression >
       void operator=(const Expr<Right_Expression>& s)  {
           for(int i=0;i<length;++i)
               v[i] = s.get(i);
       }
   }
```

The member functions

```cpp
inline double get(int i) const;
```

have to be defined as inline functions.
Difficulty of Classical Expression Templates

- At least 8 different implementations of the operator + are needed:
  
  (vector, expr), (expr, vector), (expr, expr), (vector, vector)
  (vector, double), (expr, double), (double, expr), (double, vector)

- Long template construction of the operator +:

  ```cpp
  Expr<Sum<Expr<A>,Expr<B> > > operator+ (const Expr<A>& a,
                                         const Expr<B>& b);
  ```
Barton-Nackman-Trick:

```
template <class Child>
struct Parent{
    const Child& to() const {
        return static_cast<const Child&> (*this);} 
};

class Son : public Parent<Son> ...
```

The idea of this trick is to remove the base class Parent by the member function `operator()`.
Easy Implementation of
with Barton-Nackman Trick

```
template <class A>
struct Expr {
    operator const A&() const {
        return *static_cast<const A*>(this);
    }
};

template <class A, class B>
Sum<A,B>
operator+ (const Expr<A>& a, const Expr<B>& b) {
    return Sum<A,B> (a, b);
}
```

Only 3 implementation of the operator + are needed!
template <class A, class B>
class Sum : public Expr<Sum<A,B> > {
   const A& a;
   const B& b;
public:
   Sum (const A& a_, const B& b_) : a(a_), b(b_) {}
   double get(int i) const {
      return a.get(i) + b.get(i);  
   }
};

class Vector : public Expr<Vector> {
   int n;
   double *data;
public:
   Vector ( … ) { … }
   …
   double get(int i) const { return data[i]; }

   template <class A>
   void operator= (const Expr<A> &a_){
      const A& a(a_);
      for(int i=0; i<n; ++i)
         data[i] = a.get(i);
   }
};
Easy Expression Templates

- easy use of expression templates
- extendable expression templates
- short expression template code
- We teach expression templates in a bachelor program of computational engineering.
Further ET Concepts

- return type minimization
- specialization of the operator =
- automatic parallelization with MPI and OpenMP
- fast expression templates (for vector machines)
- storage of expression templates
Colsamm is a library for the calculation of local stiffness matrices. In the library Colsamm expression templates are used at three stages:

- Description of basis functions on reference element (linear, quadratic, …)
- Description of the mapping from the reference element to the element (linear mapping or isoparametric mapping, …).
- Description of the bilinear form of the weak equation.
ET 1 in Colsamm

- Mapping of the reference element to the FE element

```cpp
Define_Element_Transformation
(
    P_0() +
    (P_1() - P_0()) * _U() +
    (P_2() - P_0()) * _V() +
    (P_3() - P_0()) * _W()
)
Tetrahedron_Transformation;
```

Here “Tetrahedron_Transformation” contains the type of the expression template. In this construction we apply “fast expression templates”.

- Isoparametric transformations are possible.
Linear elements on a tetrahedron:

```cpp
struct _Tetrahedron_linear :
  public _Domain_<4,D3,
   Tetrahedron_Transformation,
   interior,
   4,0,
   tetrahedron,
   Gauss2,
   double,
   STLVector>
```

One can define
- edge elements (for Maxwell’s equations),
- high order elements and
- mixed finite elements (for Stokes).
ET 2 in Colsamm

Description of basis functions on a reference element:

- Linear elements on a triangle:

  ```cpp
  _Tetrahedron_linear( )
  {
      this->Set( 1. – X_() – Y_() – Z() );
      this->Set( X_() );
      this->Set( Y_() );
      this->Set( Z_() );
  }
  ```

- Quadratic elements on triangle:

  ```cpp
  _Quadrangle_bilinear( )
  {
      this->Set( 4*( 1. – X_() – Y_() ) * X_() );
      this->Set( 4* X_() * Y_() );
  }
  ```
ET 3 in Colsamm

Definition of the bilinear form.

- The bilinear form corresponding to Poisson’s equation is:

\[
a(v, w) = \int_{\Omega} \nabla v \cdot \nabla w \, dz
\]

- Using expression templates this is implemented as follows:

```cpp
Triangle my_element;
std::vector<double> corners(6, 0.);
std::vector<std::vector<double>> stencil_a;
...  // set corners of the triangle
my_element(corners);

stencil_a =
    my_element.integrate(\grad(v()) \* \grad(w_()));
```
Interface in a Code for the Numerical solution of PDE`s

Library for PDE solvers
- Discretization grid
- Solver for algebraic equations
- Operators
- Parallel

Complex application code:
- Written in a mathematical language
Applications of ET in PDE Codes

The expression template concepts have to depend on the discretization and the type of the discretization grid:

- FD discretization: ET and operators for: N, S, …
  (library POOMA, Blitz++),
- FD discretization: library on staggered grids.
- FE: discretization: apply ET to the calculation of local stiffness matrices
  (Pietro and Veneziani, Joachim Härdtlein: Colsamm)
- FE: discretization: apply ET to FE-operators, restriction and prolongation operators
- FV: discretization ???
Expression Templates for Finite Elements

- Operators for FE-difference operators:

```c
F = Laplace_FE(U);
F = DX_FE(U);
```

- Elementary operators like scalar product:

```c
S = product(u,f);
```

- Combination of geometric and algebraic objects:

```c
Boundary_Neumann = ... ;
FE_space = Boundary_Neumann || interior_points;
F = Laplace_FE(U) + k * Int_boundary(U) | space_FE;
```

The last operator corresponds to the bilinear form:

\[ a(u, v) = \int_{\Omega} \nabla u \nabla v \, dz + k \int_{\Gamma_N} uv \, dz \]
Application of ET to Finite Differences

- The finite difference operator

\[ f_M = \Delta u(M) = \frac{1}{4}(u_N + u_S + u_E + u_W - 4u_M) \]

- is implemented as follows:

\[ F = 0.25 \times (N(U)+S(U)+E(U)+W(U)-4.0\times M(u)) \]
The equation \( \frac{dE}{dt} \varv = \nabla \times H \) is discretized as follows:

\[
\frac{E_{x}^{t_{n+1/2}}(M) - E_{x}^{t_{n-1/2}}(M)}{\tau} \varv = \frac{H_{z}^{t_{n}}(N) - H_{z}^{t_{n}}(S)}{h} - \frac{H_{y}^{t_{n}}(T) - H_{y}^{t_{n}}(D)}{h}
\]
FDTD for Maxwell’s Equations

- The finite difference equation

\[
E_{x}^{t_{n+1/2}}(M) = E_{x}^{t_{n-1/2}}(M) + \frac{\tau}{\varepsilon h} (H_{z}^{t_{n}}(N) - H_{z}^{t_{n}}(S) - H_{y}^{t_{n}}(T) + H_{y}^{t_{n}}(D))
\]

- is implemented as follows:

```
Variable<Dcomplex, not_staggered, staggered, staggered> Ex(grid);
Variable<Dcomplex, not_staggered, staggered, not_staggered> Hy(grid);
Variable<Dcomplex, not_staggered, not_staggered, staggered> Hz(grid);

Ex_new = Ex + tau / eps / h * ( N(Hz) - S(Hz) - T(Hy) + D(Hy) );
```
Now the expression of long vectors

\[ d = a + b + c; \]

is evaluated as follows:

```c
for(int i=0; i<length; ++i) {
    d[i] = a[i] + b[i] + c[i];
}
```
Parallelization with ET

- **Shared memory parallelization:**

  ```cpp
  template <class A>
  void Vector::operator= (const Expr<A> &a_){
    const A& a(a_);
    #pragma omp parallel for
    for(int i=0; i<n; ++i)
      data[i] = a.get(i);
  }
  ```

- **MPI parallelization:**
  Store update information in every variable.
  Do necessary update of variables, before the evaluation of the expression.

  ```cpp
  F = Laplace_FE(u) + g;
  ```
“Aliasing” Problems with ET

- Basetti F, Davis K, Quinlan D:
  - C++ Expression Templates Performance Issues in Scientific Computing

- For example,

\[
\begin{align*}
  c & = a + b * a; \\
  \text{Expr<Add<Vector,Expr<Mult<Vector,Vector> > >} & > > > \\
  \text{c.data[i] = a.data[i] + b.data[i] * a.data[i];} \\
  \text{but reaches the performance of} \\
  \text{c.data[i] = z.data[i] + y.data[i] * ž.data[i];}
\end{align*}
\]
Fast Expression Templates

- Fast Expression Templates:
  - Enumeration of vectors by template integer
  - Declare all data and functions as static
  - Call evaluation on expression types

- Changes for users
  - In some cases: template enumeration of all variables
Fast Expression Templates

- Easy Expression Templates:

```cpp
Vector a(n);
Vector b(n);
Vector c(n);

a = b*b + c;
```

- Fast Expression Templates:

```cpp
FastVector<1> a(n);
FastVector<2> b(n);
FastVector<3> c(n);

a = b*b + c;
```
Fast Expression Templates

- Easy Expression Templates:

```cpp
template <class A>
struct Expr {
    operator const A&() const {
        return *static_cast<const A*>(this);
    }
};
```

- Fast Expression Templates:

```cpp
template <class A>
struct FastExpr {};
```
Fast Expression Templates

- Easy Expression Templates:

```cpp
template <class A, class B>
Sum<A,B>
operator+ (const Expr<A>& a, const Expr<B>& b){
    return Sum<A,B> (a,b);
}
```

- Fast Expression Templates:

```cpp
template <class A, class B>
inline FastSum<A,B>
Operator+ (const FastExpr<A>& a, const FastExpr<B> b){
    return FastSum<A,B>();
}
```
Fast Expression Templates

• Easy Expression Templates:

```cpp
template <class A, class B>
class Sum : public Expr < Sum<A,B> > {
    const A& a;
    const B& b;
    public:
        Sum (const A& a_, const B& b_) : a(a_), b(b_) {}  
        double get(int i) const {
            return a.get(i) + b.get(i);}
}
```

```cpp
template <class A, class B> Struct FastSum : public FastExpr < Sum<A,B> > {
    static double get(int i) const {
        return A::get(i) + B::get(i); }
}
```
Fast Expression Templates

- Fast Expression Templates:

```
Template<int i>
class FVector : public FastExpr<FVector<int> > {
    int n;
    static double *data;
public:
    Vector ( ... ) { ... }
    ...
    double get(int i) const { return data[i]; }

    template <class A>
    void operator= (const FastExpr<A> &a_){
        for(int i=0; i<n; ++i)
            data[i] = E::get(i);
    }
};
```

```
Template<int id> double* FVector<int>::data;
```
Automatic Parallelization with ET

- The following code runs in serial and parallel in EXPDE:

```c
u = 0.0 | points_cooling; // for Dirichlet-Neumann
u = 0.0 | points_in_space; // boundary
f = f | points_in_space;   // conditions
g = Laplace_FE(u) - f;    // cg-iteration
d = -g;
delta = product(g,g);
for(int i=1;i<=k;++i) {
    r = Laplace_FE(d);
    tau = delta / product(d,r);
    u = u + tau * d;
    g = g + tau * r;
    delta_prime = product(g,g);
    beta = delta_prime / delta;
    delta = delta_prime;
    d = beta * d - g;
}
```

- Every variable stores an update information.
- A “parallel update” is performed of the expression requires an update.
NEC SX 6 vector machine:

Differences

for(int i = 0; i < iter_max; ++i){
    d = 4*h2*b + North(u) + South(u) +
        East(u) + West(u);
    u = d;
}